**Explain TCP / IP**

The transport layer protocols are TCP and UDP. They use the n/w layer protocol IP which is either IPv4 or IPv6. It is possible to use IPv4 or IPv6 directly, bypassing the transport layer, this technique, often called *raw sockets*, is used much less frequently.

**UDP**

UDP is a simple transport-layer protocol. The application writes a message to a UDP socket, which is then *encapsulated* in a UDP *datagram*, and sent to its destination.

There is **no reliability**, If a datagram reaches its final destination but the checksum detects an error, or if the datagram is dropped in the network, it is not delivered to the UDP socket and is not automatically retransmitted.

**Datagram** because the message has clear boundaries.

Each UDP datagram has a length. The length of a datagram is passed to the receiving application along with the data.

**We also say that UDP provides a *connectionless* service**, as there need not be any long-term relationship between a UDP client and server. For example, a UDP client can create a socket and send a datagram to a given server and then immediately send another datagram on the same socket to a different server. Similarly, a UDP server can receive several datagrams on a single UDP socket, each from a different client.

**TCP**

TCP is a connection oriented protocol. It sets up *connections* between clients and servers. A TCP client establishes a connection with a given server, exchanges data with that server across the connection, and then terminates the connection.

**TCP also provides *reliability***. When TCP sends data to the other end, it requires an

acknowledgment in return. If an acknowledgment is not received, TCP automatically

retransmits the data and waits a longer amount of time.

Note that TCP does not guarantee that the data will be received by the other endpoint, as this is impossible. It delivers data to the other endpoint if possible, and notifies the user if it is not possible. Therefore, TCP cannot be described as a 100% reliable protocol; it provides reliable delivery of data *or* reliable notification of failure.

**TCP estimate the *round-trip time* (RTT)** between a client and server dynamically so that it knows how long to wait for an acknowledgment.

**TCP also *sequences***the data by associating a sequence number with every byte that it

sends (A *segment* is the unit of data that TCP passes to IP.) If the segments arrive out of order, the receiving TCP will reorder the two segments based on their sequence numbers before passing the data to the receiving application. If TCP receives duplicate data from its peer it can detect that the data has been duplicated and discard the duplicate data.

**TCP provides *flow control***. TCP always tells its peer exactly how many bytes of data it is

willing to accept from the peer at any one time. This is called the advertised *window*. At any time, the window is the amount of room currently available in the receive buffer, guaranteeing that the sender cannot overflow the receive buffer. The window changes dynamically over time: As data is received from the sender, the window size decreases, but as the receiving application reads data from the buffer, the window size increases. It is possible for the

window to reach 0: when TCP's receive buffer for a socket is full and it must wait for the

application to read data from the buffer before it can take any more data from the peer.

UDP provides no flow control. It is easy for a fast UDP sender to transmit datagrams at a rate that the UDP receiver cannot keep up with, as we will show in Section 8.13.

**Finally, a TCP connection is *full-duplex***. This means that an application can send and receive data in both directions on a given connection at any time.

UDP can be full-duplex.

**TCP Connection Establishment and Termination**

**TCP connection is a Three-Way Handshake**

The following scenario occurs when a TCP connection is established:

The server must be prepared to accept an incoming connection. This is normally done

by calling socket, bind, and listen and is called a ***passive open***.

1. The client issues an *active open* by calling connect. This causes the client TCP to

send a **"synchronize" (SYN) segment**, which tells the server the client's initial sequence number for the data that the client will send on the connection. Normally, there is no data sent with the SYN; **it just contains an IP header, a TCP header, and possible TCP options (which we will talk about shortly).**

2. The server must acknowledge (ACK) the client's SYN and the server must also send its

own SYN containing the initial sequence number for the data that the server will send

on the connection. The server sends its SYN and the ACK of the client's SYN in a

single segment.

3. The client must acknowledge the server's SYN.

The minimum number of packets required for this exchange is three; hence, this is called TCP's *three-way handshake*. We show the three segments in Figure 2.2.



Where J is the starting sequence number of the client and K is starting sequence number of the Server.

**TCP Options**

Each SYN can contain TCP options. Commonly used options include the following:

**MSS option**. With this option, the TCP sending the SYN announces its ***maximum***

***segment size***, the maximum amount of data that it is willing to accept in each TCP

segment, on this connection. The sending TCP uses the receiver's MSS value as the

maximum size of a segment that it sends.

**Window scale option**. The maximum window that either TCP can advertise to the other TCP is 65,535, because the corresponding field in the TCP header occupies 16 bits.

But, high-speed connections, common in today's Internet or long delay paths

(satellite links) require a larger window to obtain the maximum throughput possible.

This newer option specifies that the advertised window in the TCP header must be

scaled (left-shifted) by 0-14 bits, providing a maximum window of almost one gigabyte

(65,535 x 214). Both end-systems must support this option for the window scale to be

used on a connection.

**TCP Connection Termination**

While it takes three segments to establish a connection, it takes four to terminate a

connection.

1. One application calls close first, and we say that this end performs the *active close*.

This end's TCP sends a ("Final") FIN segment, which means it is finished sending data.

2. The other end that receives the FIN performs the passive close. The received FIN is

acknowledged by TCP. The receipt of the FIN is also passed to the application as an

end-of-file (after any data that may have already been queued for the application to

receive), since the receipt of the FIN means the application will not receive any additional data on the connection.

3. Sometime later, the application that received the end-of-file will close its socket. This causes its TCP to send a FIN.

4. The TCP on the system that receives this final FIN (the end that did the active close)

acknowledges the FIN.

Since a FIN and an ACK are required in each direction, four segments are normally required.



A FIN occupies one byte of sequence number space just like a SYN, The ACK of

each FIN is the sequence number of the FIN plus one.

**Practical Scenarios when the close gets called (**written in detail later**)**

**- When the end performing passive close is in middle of sending data**

Between Steps 2 and 3 it is possible for data to flow from the end doing the passive close to the end doing the active close. This is called a *half-close* and we will talk about this in detail with the shutdown function in Section 6.6.

**- Close initiated by the TCP when the process terminates voluntarily or involuntarily**

The sending of each FIN occurs when a socket is closed. We indicated that the application calls close for this to happen, but realize that when a Unix process terminates, either voluntarily (calling exit or having the main function return) or involuntarily (receiving a signal that terminates the process), all open descriptors are closed, which will also cause a FIN to be sent on any TCP connection that is still open.

**Buffer Sizes and Limitations**

**TCP Output**



**Write from application buffer to the send buffer**

Every TCP socket has a send buffer and we can change the size of this buffer with the

SO\_SNDBUF socket option (Section 7.5). When an application calls write, the kernel copies all the data from the application buffer into the socket send buffer. If there is insufficient room in the socket buffer for all the application's data (either the application buffer is larger than the socket send buffer, or there is already data in the socket send buffer), **the process is put to sleep.** This assumes the normal default of a blocking socket. (We will talk about nonblocking sockets in Chapter 16.) The kernel will not return from the write until the final byte in the application buffer has been copied into the socket send buffer. **Therefore, the successful return from a write to a TCP socket only tells us that we can reuse our application buffer.** It does *not* tell us that either the peer TCP has received the data or that the peer application has received the data. (We will talk about this more with the SO\_LINGER socket option in

Section 7.5.)

TCP takes the data in the socket send buffer and sends it to the peer TCP based on all the rules of TCP data transmission. The peer TCP must

acknowledge the data, and as the ACKs arrive from the peer, only then can our TCP discard the acknowledged data from the socket send buffer. TCP must keep a copy of our data until it is acknowledged by the peer.

**Send Buffer -> IP -> Data Link Queue -> Peer TCP**

TCP sends the data to IP in MSS-sized (maximum segment sized) or smaller chunks, prepending its TCP header to each segment, where the MSS is the value announced by the peer, or 536 if the peer did not send an MSS option. IP prepends its header, searches the routing table for the destination IP address (the matching routing table entry specifies the outgoing interface), and passes the datagram to the appropriate datalink. IP might perform fragmentation before passing the datagram to the datalink, but as we said earlier, one goal of the MSS option is to try to avoid fragmentation and newer implementations also use path MTU discovery. Each datalink has an output queue, and if this queue is full, the packet is discarded and an error is returned up the

protocol stack: from the datalink to IP and then from IP to TCP. TCP will note this error and try sending the segment later. The application is not told of this transient condition.

**UDP Output**

The socket send buffer does not exist for UDP socket, so we are showing it as dashed box. A UDP socket has a send buffer size (which we can change with the SO\_SNDBUF socket option, Section 7.5), but this is simply an upper limit on the maximum-sized UDP datagram that can be written to the socket. If an application writes a datagram larger than the socket send buffer size, EMSGSIZE is returned. Since UDP is unreliable, it does not need to keep a copy of the application's data and does not need an actual send buffer. (The application data is normally copied into a kernel buffer of some form as it passes down the protocol stack, but this copy is discarded by the datalink layer after the data is transmitted.)

UDP simply prepends its 8-byte header and passes the datagram to IP. IPv4 or IPv6 prepends its header, determines the outgoing interface by performing the routing function, and then either adds the datagram to the datalink output queue (if it fits within the MTU) or fragments the datagram and adds each fragment to the datalink output queue. If a UDP application sends large datagrams (say 2,000-byte datagrams), there is a much higher probability of fragmentation than with TCP, because TCP breaks the application data into MSS-sized chunks, something that has no counterpart in UDP.

**The successful return from a write to a UDP socket tells us that either the datagram or all fragments of the datagram have been added to the datalink output queue. If there is no room on the queue for the datagram or one of its fragments, ENOBUFS is often returned to the application.**

Unfortunately, some implementations do not return this error, giving the application no

indication that the datagram was discarded without even being transmitted.

**TCP / IP Protocol**

****

**Generic Socket Options**

Below are the important or commonly used socket options

**SO\_BROADCAST Socket Option**

This option enables or disables the ability of the process to send broadcast messages.

Broadcasting is supported for only datagram sockets and only on networks that support the concept of a broadcast message (e.g., Ethernet, token ring, etc.). You cannot broadcast on a point-to-point link or any connection-based transport protocol such as SCTP or TCP.

**SO\_KEEPALIVE Socket Option**

When the keep-alive option is set for a TCP socket and no data has been exchanged **across the socket in either direction for two hours, TCP automatically sends a *keep-alive probe* to the peer**. This probe is a TCP segment to which the peer must respond. One of three scenarios results:

1. The peer responds with the expected ACK. The application is not notified (since

everything is okay). TCP will send another probe following another two hours of

inactivity.

2. The peer responds with an RST, which tells the local TCP that the peer host has

crashed and rebooted. The socket's pending error is set to ECONNRESET and the socket

is closed.

3. There is no response from the peer to the keep-alive probe. Berkeley-derived TCPs send 8 additional probes, 75 seconds apart, trying to elicit a response. TCP will give up if there is no response within 11 minutes and 15 seconds after sending the first probe. If there is no response at all to TCP's keep-alive probes, the socket's pending error is set to ETIMEDOUT and the socket is closed.

The purpose of this option is to detect if the peer *host* crashes or becomes unreachable (e.g., dial-up modem connection drops, power fails, etc.). **If the peer *process* crashes, its TCP will send a FIN across the connection, which we can easily detect with select.** (This was why we used select in Section 6.4.) Also realize that if there is no response to any of the keep-alive probes (scenario 3), we are not guaranteed that the peer host has crashed, and TCP may well terminate a valid connection( It could be that some intermediate router has crashed).

This option is normally used by servers, although clients can also use the option. Servers use the option because they spend most of their time blocked waiting for input across the TCP connection, that is, waiting for a client request. But if the client host's connection drops, is powered off, or crashes, the server process will never know about it, and the server will continually wait for input that can never arrive. This is called a *half-open connection*. The keep-alive option will detect these half-open connections and terminate them.

****

**Ways to detect various TCP conditions.**

**SO\_LINGER Socket Option**

This option specifies how the close function operates for a connection-oriented protocol

(e.g., for TCP and SCTP, but not for UDP). By default, close returns immediately, but if there is any data still remaining in the socket send buffer, the system will try to deliver the data to the peer.

The SO\_LINGER socket option lets us change this default.

**SO\_RCVBUF and SO\_SNDBUF Socket Options**

Every socket has a send buffer and a receive buffer. We described the operation of the send buffers with TCP, UDP, and SCTP in Figures 2.15, 2.16, and 2.17.

The receive buffers are used by TCP, UDP, and SCTP to hold received data until it is read by the application. With TCP, the available room in the socket receive buffer limits the window that TCP can advertise to the other end. The TCP socket receive buffer cannot overflow because the peer is not allowed to send data beyond the advertised window. This is TCP's flow control, and if the peer ignores the advertised window and sends data beyond the window, the receiving TCP discards it. With UDP, however, when a datagram arrives that will not fit in the socket receive buffer, that datagram is discarded. Recall that UDP has no flow control: It is easy for a fast sender to overwhelm a slower receiver, causing datagrams to be discarded by the receiver's UDP, as we will show in Section 8.13. In fact, a fast sender can overwhelm its own network interface, causing datagrams to be discarded by the sender itself.

These two socket options let us change the default sizes.

**TCP Socket Options**

**TCP\_MAXSEG Socket Option**

This socket option allows us to fetch or set the MSS for a TCP connection. The value

returned is the maximum amount of data that our TCP will send to the other end; often, it is the MSS announced by the other end with its SYN, unless our TCP chooses to use a smaller value than the peer's announced MSS.

**TCP\_NODELAY Socket Option**

If set, this option disables TCP's *Nagle algorithm* (Section 19.4 of TCPv1 and pp. 858 859 of TCPv2). By default, this algorithm is enabled.

The purpose of the *Nagle algorithm* is to reduce the number of small packets on a WAN. The algorithm states that if a given connection has outstanding data (i.e., data that our TCP has sent, and for which it is currently awaiting an acknowledgment), then no small packets will be sent on the connection in response to a user write operation until the existing data is acknowledged.

**Select**

**Under What Conditions Is a Descriptor Ready?**

1. A socket is ready for reading if any of the following four conditions is true:

a. The number of bytes of data in the socket receive buffer is greater than or

equal to the current size of the low-water mark for the socket receive buffer. A

read operation on the socket will not block and will return a value greater than

0 (i.e., the data that is ready to be read). We can set this low-water mark

using the SO\_RCVLOWAT socket option. The default value is 1 (1 byte)for TCP and UDP sockets.

b. The read half of the connection is closed (i.e., a TCP connection that has

received a FIN). A read operation on the socket will not block and will return 0

(i.e., EOF).

c. The socket is a listening socket and the number of completed connections is

nonzero. An accept on the listening socket will normally not block, although we

will describe a timing condition in Section 16.6 under which the accept can

block.

d. A socket error is pending. A read operation on the socket will not block and will

return an error (-1) with errno set to the specific error condition.

2. A socket is ready for writing if any of the following four conditions is true:

a. The number of bytes of available space in the socket send buffer is greater

than or equal to the current size of the low-water mark for the socket send

buffer *and* either: (i) the socket is connected, or (ii) the socket does not

require a connection (e.g., UDP). The low-water mark normally defaults to 2048 for TCP and UDP sockets.

b. The write half of the connection is closed. A write operation on the socket will

generate SIGPIPE (Section 5.12). Write on a closed write half buffer returns SIGPIPE signal.

c. A socket using a non-blocking connect has completed the connection, or the

connect has failed.

d. A socket error is pending. A write operation on the socket will not block and will return an error (-1) with errno set to the specific error condition.

**Summary of conditions that cause a socket to be ready for**

**select.**



**TCP Client/Server termination condition**

**Termination of server process**

- When the server process crashes or if it is killed, it calls the close on the socket. This results in the "Finalize" FIN being sent to the client. Client acknowledges the FIN.

- The client socket is ready for the read and if the read is performed on the socket it returns 0 indicating EOF. The client can now also call the close.

- If the client is in the middle of sending data, the first write copies the data from the application buffer to the TCP buffer. The TCP then sends this data to the Server which then replies back with RST("reset"). This will result in write buffer close and any further write will result in SIGPIPE signal being generated. This signal should be handled by the process or the process will be terminated. In the case where the process is programmed to ignore the SIGPIPE by setting the SIG\_IGN for SIGPIPE, this signal is ignored and the write returns with error EPIPE. The client can now close the connection.

**Crashing of server host**

The server crashes and is down or the n/w cable is unplugged.

- When the server host crashes nothing is sent out on the existing network connections.

- If the client writes to the socket the data is copied from the user buffer to the TCP Buffer and the tcp then tries to send the data to the client.

- The client tcp continuously transmits the data to the server and waits for ACK. The client tcp tries to send data segment 12 times and waits for around 9 minutes to receive the ACK. It finally gives up and sets the error as 'ETIMEDOUT' or 'EHOSTUNREACH' if the router in the path responds with the "unreachable host"

- Any further read or write will result in error (-1) and errno set.

**Crashing and rebooting of server host**

In this case the server crashes and reboots

- The server and the client are both brought up and connected, we also test the connection by sending the test data from the client to the server.

- The server host is crashed and rebooted when it reboots it losses all the information about the TCP connection. When the client sends any data to the server the server does not recognises the connection and replies back with RST ("reset").

- If the client calls the write after the 'RST' it will error out with SIGPIPE.

- If the client calls the read this will error out with error ECONNRESET.

**A very good example of when to use WriteFds in select**

<https://stackoverflow.com/questions/20071865/why-use-writefds-in-select-how-to-use-them-in-practice>

The writefds of select is to check that the file descriptor is ready for writing. For a socket that means that the send buffer associated with the socket is not full.

Let's assume that the sockets on your platform have an 8 kb buffer and you want to send 100 kb of data.

You call write and get a return value of 8192 indicating that the first 8192 bytes have been written. The next call to write returns either EAGAIN or EWOULDBLOCK indicating that the send buffer is full.

**You can now use select to find out when there is room in send buffer again** (that is, when one tcp/ip packet has been transferred to the client) so that you can continue writing. At the same time you can be listening for new connections and waiting for input from clients.

For most of the practical use of select it is used with **WriteFds as NULL** as in this example.

int

main(void) {

fd\_set rfds;

struct timeval tv;

int retval;

/\* Watch stdin (fd 0) to see when it has input. \*/

FD\_ZERO(&rfds);

FD\_SET(0, &rfds);

/\* Wait up to five seconds. \*/

tv.tv\_sec = 5;

tv.tv\_usec = 0;

**retval = select(1, &rfds, NULL, NULL, &tv);**

/\* Don’t rely on the value of tv now! \*/

if (retval == -1)

perror("select()");

else if (retval)

printf("Data is available now.\n");

/\* FD\_ISSET(0, &rfds) will be true. \*/

else

printf("No data within five seconds.\n");

return 0;

}

**int select(int** *nfds***, fd\_set \****readfds***, fd\_set \****writefds***,**

**fd\_set \****exceptfds***, struct timeval \****timeout***);**

**void FD\_CLR(int** *fd***, fd\_set \****set***);**

**int FD\_ISSET(int** *fd***, fd\_set \****set***);**

**void FD\_SET(int** *fd***, fd\_set \****set***);**

**void FD\_ZERO(fd\_set \****set***);**

**FD\_ISSET(int** *fd***, fd\_set \****set***);**

The inputs are fd and set, fd is the filedescriptor we are interested in and the set is the one which has captured the values of things which we are interested in.

**Shut down of Server host**

If the server host is shut down it will send the SIGTERM signal to all the process and then the SIGKILL. When the processes are terminated it will close the open descriptors. This will result in FIN to be sent to the client which can then identify this by select and close the client connection.

**Conditions handled by select for readability.**



Three conditions are handled with the socket:

1. If the peer TCP sends data, the socket becomes readable and read returns greater

than 0 (i.e., the number of bytes of data).

2. If the peer TCP sends a FIN (the peer process terminates), the socket becomes

readable and read returns 0 (EOF).

3. If the peer TCP sends an RST (the peer host has crashed and rebooted), the socket

becomes readable, read returns 1, and errno contains the specific error code.

**Non Blocking I/O**

**Memory organization of a typical C/C++ program**



**1. Code segment or text segment:** Code segment contains the code executable or code binary.

**2. Data segment:** Data segment is sub divided into two parts

– Initialized data segment: All the global, static and constant data are stored in the data segment.

– Uninitialized data segment: All the uninitialized data are stored in **BSS**.  
**3. Heap**: When program allocate memory at runtime using calloc and malloc function, then memory gets allocated in heap. when some more memory need to be allocated using calloc and malloc function, heap grows upward as shown in above diagram.

**4. Stack**: Stack is used to store your local variables and is used for passing arguments to the functions along with the return address of the instruction which is to be executed after the function call is over. When a new stack frame needs to be added (as a result of a newly called function), the stack grows downward.

**Explain UDP**

**When to use UDP vs TCP**

**Adding functionalities to UDP stack**